An Open64-based Compiler Approach to Performance Prediction and Performance Sensitivity Analysis for Scientific Codes

Jeremy Abramson
University of Southern California / Information Sciences Institute
4676 Admiralty Way*, Suite 1001*
Marina del Rey, Ca 90292
abramson@isi.edu

Pedro C. Diniz
Instituto Superior Técnico / INESC-ID
Tagus Park
2780-990 Oeiras
Portugal
pedro.diniz@tagus.ist.utl.pt

Abstract

The lack of tools that provide performance feedback at a level of abstraction programmers can relate to makes the problem of performance prediction and portability extremely difficult. Programmers have no simple way of knowing what programming constructs significantly effect performance in today's machines, much less in machines that are under development and are not readily available. We describe an Open64-based compiler approach to the problem of performance prediction and architecture sensitivity analysis done at a source-level. Our analysis tool extracts the computation's high-level dataflow-graph from the code's WHIRL representation, and uses source-level data access patterns information as well as register needs to derive performance bounds for the program under various architectural scenarios. The end result is a very fast performance prediction as well as insight into where performance bottlenecks are. We have experimented with a real code engineers and scientists use in practice – a sparse matrix-vector multiplication kernel. The results correlate very well with the execution of the code on a real machine and allow programmers to understand the performance bottlenecks without having to engage in very low-level instrumentation analysis.

Categories and Subject Descriptors C.4.3 [**Performance of Systems**]: Measurement techniques, modeling techniques, performance attributes

General Terms: Measurement, Performance.

Keywords *performance prediction and modeling; program analysis; architecture modeling; Open64*

1. Introduction and Background

Modern high-end computers present a complex execution environment that makes performance understanding and performance portability extremely difficult. Programmers go to extreme lengths to manually apply various high-level transformations (most notably loop-unrolling) in an attempt to expose more Instruction-Level Parallelism (ILP). Exploiting ILP allows the compiler to take advantage of micro architecture features such as pipelining, superscalar and multi-core characteristics. The lack of performance prediction and analysis tools that can provide meaningful performance feedback leave the programmer in an unfortunate situation. Without understanding why the performance is what it is, the programmer is forced to search for the best possible transformation sequences by trial and error. At best, existing performance understanding tools provide feedback at a very low level of abstraction – such as cache miss rates – that provide no clue as to what architectural bottlenecks, if any, lead to such results.

Earlier approaches to performance modeling and understanding were purely empirical. Researchers developed representative kernel codes of large-scale applications such as the NAS Parallel [8] and the SPEC [9] benchmarks. By observing the performance of these kernels on a given machine one could extrapolate in a qualitative fashion the performance behavior of a real application. More recently researchers have developed models for the performance of parallel applications by examining their memory behavior [5,6]. Other work has focused on modeling the behavior of an application by first accurately characterizing the running time of the sequential portions of the application. This is done by using analytical modeling, based on intimate knowledge of the application's mathematics, in addition to empirical observations to extract the corresponding parameter values [7]. On the other end of the spectrum, cycle-level simulators for architecture performance understanding at a very low level are simply too slow for realistic workloads. As a result the simulations tend to focus on a minute subset of the instruction stream or use sampling techniques and are thus limited to very focused architectural analyses.

This paper describes an alternative approach to the problem of performance prediction and architecture sensitivity analysis using source level program analysis and scheduling techniques. An architectural overview of the tool is given in Figure 1. Our approach looks at the High-Level (HL) WHIRL tree and isolates the basic blocks of the input source program. It then extracts the corresponding high-level data-flow graph (DFG) information. Using high-level information about the data access patterns of array references, it determines the expected latency of memory operations. Once the DFG of each basic block - most notably the ones in the body of nested loops - is extracted, the compiler tool uses a list-scheduling algorithm to determine the execution time of the computation. This scheduling makes use of the DFG as well as a number of other tunable architecture parameters, such as the

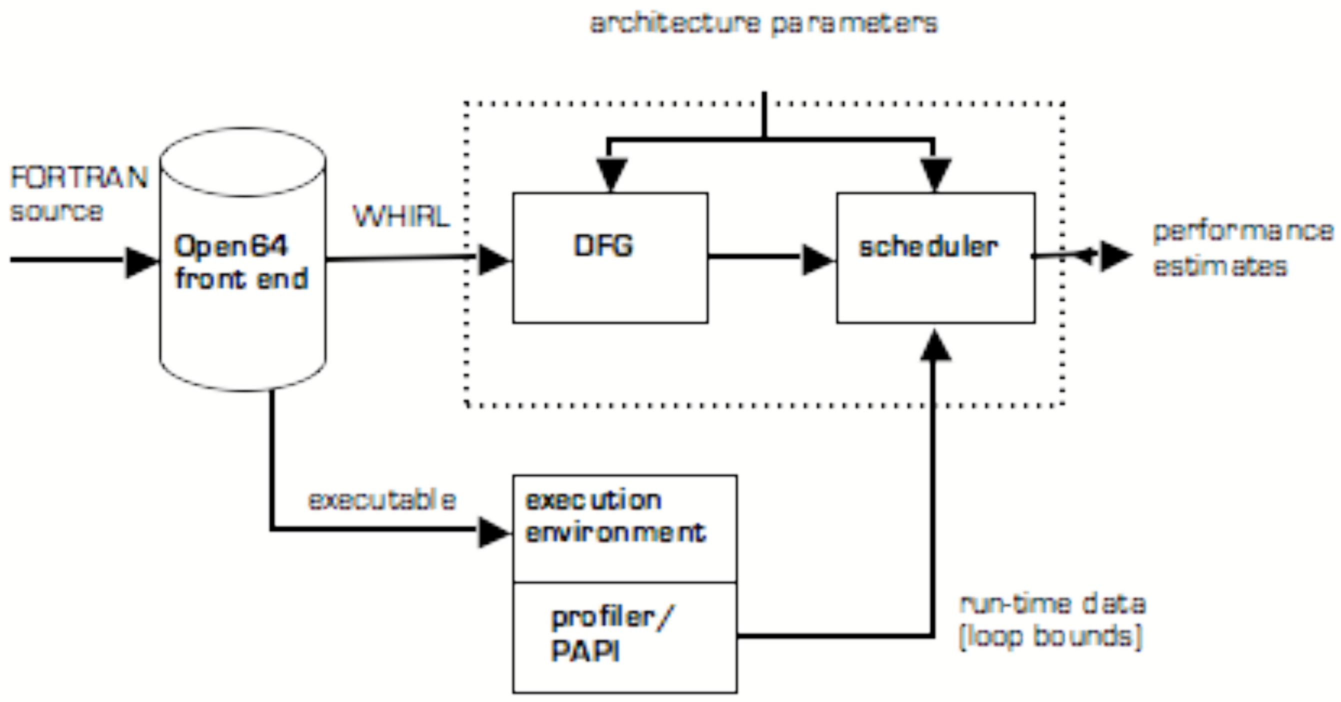

Figure 1. Analysis tool architectural block diagram

number of load/store or arithmetic units, the number of available registers and specific operation latency values.

We have developed our tool based on the Open64 compiler infrastructure [3], and experimented with this approach using the CG sparse matrix-vector multiplication kernel, a synthetic kernel from the original NAS benchmark set. Using this computational kernel our tool determines qualitatively that in the absence of loop unrolling no more than three functional load/store units are needed to attain a level of performance that is consistent with the critical path of the computation (see figure 5). When the core is unrolled by a factor of 8 no more than four functional load/store units are needed (figure 6). In the context of a multi-core architecture, this information would allow a compiler to adjust the concurrency in the generated code to match the target architecture resources, in some cases even by dynamically adapting its run-time execution strategy to unroll just the required amount depending on the available units.

The remainder of this paper is organized as follows. In the next section we describe in more detail the technical approach of our tool and how it provides performance predictions and architectural sensitivity analysis. Section 3 presents the experimental results for our case study application - the NAS CG kernel code. We present concluding remarks in section 4.

2. Technical Approach

We now describe the details of our technical approach to the problem of performance prediction and sensitivity analysis. Using high-level WHIRL source code information, static compiler data and control- dependence analysis techniques, we can determine performance bounds and performance sensitivity in a fraction of the time it takes to run the actual code.

2.1 Basic Analyses: Data-Flow Graph (DFG)

This analysis tool extracts the basic blocks at the source code level by inspection of the compiler intermediate representations in WHIRL. Because the front-end of the compiler does perform some deconstruction of high-level constructs, most notably while-loops, it is not always possible to map the intermediate representation constructs back to source code constructs. Despite some of these shortcomings the front-end does keep track of line number information that allows the tool to provide reasonably accurate feedback to the programmer.

For each basic block the compiler extracts a data-flow graph, accurately keeping track of dependencies (true-, anti-, input- and output-dependencies) via scalar variables and conservatively assume that any reference to an array variable may induce a dependency. In some cases, the compiler uses data dependence analysis techniques (see e.g., [4]) to disambiguate the references to arrays and thus eliminate false dependences in the DFG. In the current implementation we make the optimistic assumption that arrays with distinct symbolic names are unaliased. While this assumption is clearly not realistic in the general case, it holds for the Fortran kernel code in our controlled experimental results. Future revisions will handle cases where symbols can be aliased.

Figure 2 shows the dataflow graph resulting from the WHIRL representation of the innermost loop nest of CG. Each node in the DFG corresponds to a line of WHIRL from the abstract syntax tree. The operational latencies are given by the edge-weights of the graph (see table 2). Because of the high level intermediate representation, array references are not lowered, and can be easily reconstructed from their corresponding array nodes in the DFG. This allows the programmer to gain important insight to which operations most affect performance. The original Fortran code is shown in Figure 3, where the critical computation is performed using an indirect access to the *y* array variable.

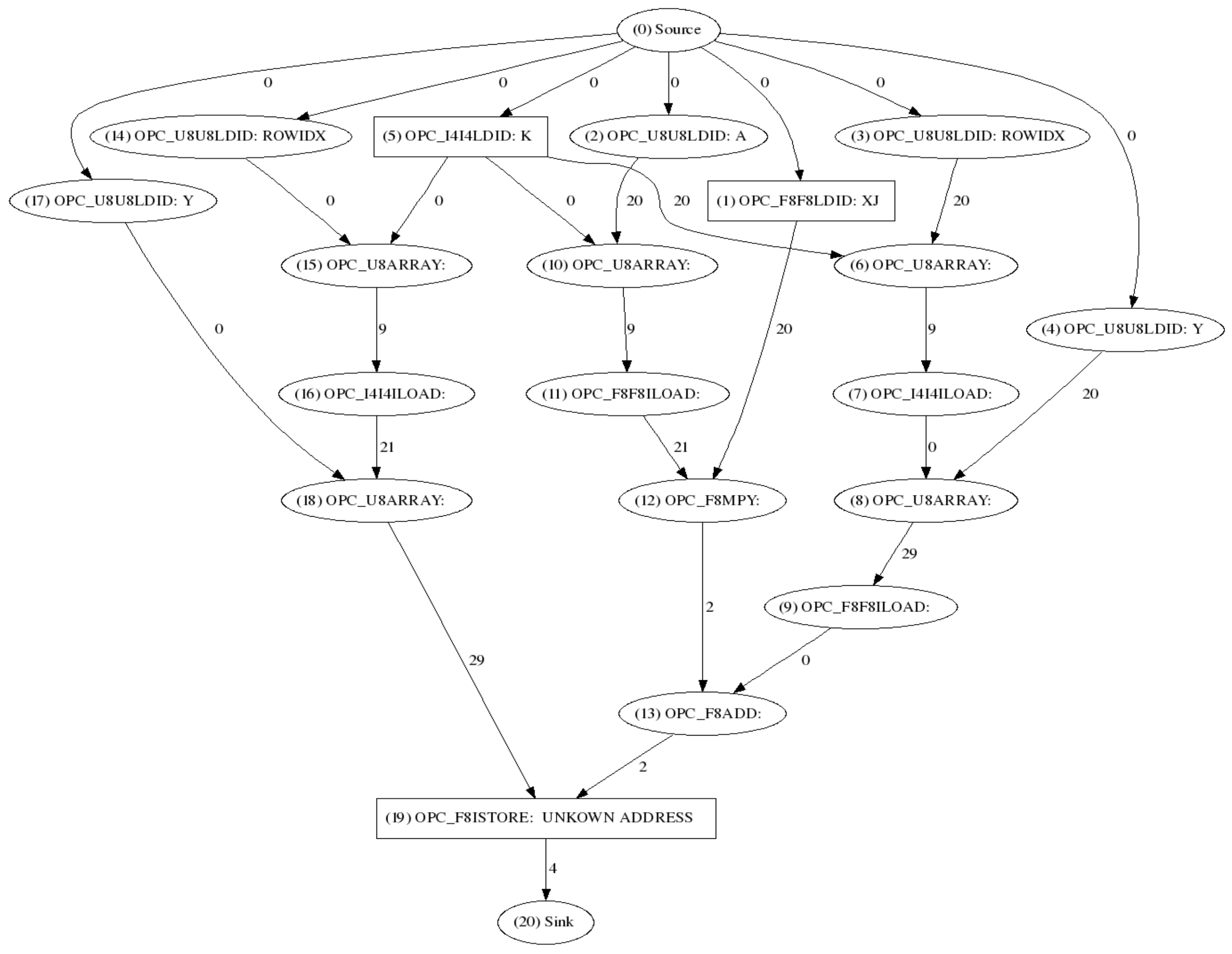

Figure 2. Data-flow graph for the core of CG

Finally, we identify the loops of the code across the various basic blocks of a procedure to uncover basic and derived induction variables. This information is vital in determining array access stride information as explained below.

2.2 Data Access Pattern Analysis

In this analysis the compiler extracts the affine relations between scalar variables in the array indexing functions, taking into account the basic and derived induction variables whenever possible. For example, knowing that scalar variables i and j are loop induction variables, the array reference a[i][j+1] has array subscripts with affine coefficients (1,0,0) and (0,1,1) respectively. The last element in each tuple corresponds to the constants 0 and 1 in the expressions i+0 and j+1. Using this access information and the layout of the array (i.e., either column-wise and row-wise) the analysis determines the stride information for each access and estimates the latency of the corresponding memory operations. We use the assumptions that regular memory access are very likely to hit the cache or reside in registers as a result of an aggressive pre-fetching algorithm [1] whereas an irregular or random memory reference is very likely to miss the cache. The construction of the DFG uses this knowledge to decorate the latency of the individual array accesses as either regular or irregular thus taking into account, to some extent, the effects of the memory hierarchy. Future work will focus on refining the array data-reuse model, adopting a strategy such as that given in [14].

2.3 Memory Hierarchy Effects

Given a DFG whose nodes are decorated with data access pattern information the next step is to assess the latency of the various load/store operations. Our analysis tackles the memory hierarchy efforts by modeling cache effects as well as register allocation for the various symbolic variables the computation manipulates. As described in Table 1 we use a cost model based on the stride and observed range of the various data references during execution. The table also includes a column to show that in cases where prefetching is being modeled, the cost of a load operation is zero at the expense of an additional register. This prefetching is not used when the data access pattern is irregular, as the address cannot be computed before the load is executed. Note that some of the range analysis requires profiling knowledge of the computation that is obtained by source-level instrumentation that is fairly lightweight as described in [10].

	Stride		Prefetched /in register
	Short (less than L1 cache line)	Long (larger than L1 cache line)	
Regular	L1 hit latency on all but first of K accesses to each line	L1 miss latency	0
Irregular	L1 miss latency	L1 miss latency	---

Table 1. Modeling Caching and Data Reuse Behavior for Array Data References (*K is ratio of cache line size over stride*)

In limited cases the compiler can uncover self- and group-reuse opportunities. These are important when exploring cache-line size reuse (also known as spatial reuse). When an array data reference exhibits reuse across iterations of an outermost loop of a nest, if the range of locations visited in a previous invocation of the innermost loops overlaps with the current invocation, then we consider all the overlapped locations to be still residing in cache as long as the reused data is smaller than the cache size (capacity reuse). When the range of reused data items is simply too big, irrespective of the type of stride, then we consider the addressed location not to reside in cache. This capacity argument is used for both L1 and L2 caches.

Finally, we use a register allocation strategy for the symbolic variables the code manipulates. The first time a symbol is referenced in the DFG, it will pay a miss penalty, as given by table 1. It then resides in a register and incurs latency of 0, until it is spilled. Both the overall number of registers and the method to choose which register to spill are modular, and can be changed to see the effect different schemes have on performance. Initially, we choose a scheme whereby the symbol with the fewest number of [future] uses is evicted from the register file. This is similar to the "top-down" register allocation algorithm given in [15].

2.4 Hardware Resources and Scheduling Analysis

We develop our own operation scheduler for determining the latency of execution of each basic block, given its DFG. In this scheduler we can program the latencies of the individual operations, taking into account the memory hierarchy effects as described above, as well as if they are executed in a pipelined fashion or not. Our scheduler also allows us to specify the number of functional units for either each individual type of operations or for a generic functional unit. For example we can segregate the arithmetic and floating-point operations in a single functional units or allow all of them to be executed in a generic functional unit with both integer and floating-point operations. We can also specify multiple load and store units thus modeling the available bandwidth of the target architecture. Finally, we assume the scheduler is an on-line as-soon-as-possible scheduling algorithm with zero-time overhead in scheduling of the various operations in the functional units.

This analysis allows us to derive upper bounds for the expected performance of the computation at an intermediate level of representation and thus provide meaningful feedback to both the programmer and the architecture designer. By considering an infinite number of registers and infinite number of functional units or memory bandwidth, the programmer can quickly determine which statements at the source-code level contribute to the critical path of the computation. These inherent dependencies thus constrain the maximum achievable performance. For more realistic scenarios the critical path is determined by the need to reload values in registers or by the limited available memory bandwidth. The approach can determine for each computation, possibly transformed at the source code level, where the performance bottlenecks are. In addition, by varying the architectural parameters, the architecture designer can quickly determine what architectural configuration can best realize performance gains. For example, a computation whose critical path is made up of many high-latency arithmetic operations (such as floating-point divide or square root) will likely realize performance improvements when run on a machine in which these operations are pipelined (versus a machine with more load/store units, for example). Our approach allows these scenarios to be quickly tested and analyzed, without having to actually run the code on such a machine.

Our scheduler, though simple, allows us to anticipate the completion time of the operations corresponding to a given basic block along with various efficiency metrics such as the number of clock cycles a given computation was stalled awaiting an available functional unit or awaiting a data dependency to be satisfied.

```
do 200 j = 1, n
        xj = x(j)
        do 100 k = colstr(j) , colstr(j+1)-1
        // DFG shown in fig. 2
                y(rowidx(k)) = y(rowidx(k)) + a(k) + xj
        100 continue
200  continue
```

Figure 3. Fortran source for the core of CG

3. A Case Study

We now present preliminary experimental results of the perform-

Load (cache miss)	Load Address	32-bit int. Add	32-bit Int. Multiply	32-bit FP Multiply	32-bit FP Divide	Array Address Calculation (Non-affine)	Array Address Calculation (Affine)
20	2	1	2	2	12	29	9

Table 2. Selected operation latencies

ance expectation and sensitivity analysis for a synthetic code, NAS CG. The NAS CG kernel is written in FORTRAN and implements the conjugate-gradient iterative refinements method for a positive-definite input sparse-matrix. At the core of this kernel is a sparse-matrix vector multiplication. We first describe the methodology followed in these experiments and then present and discuss our findings using our analysis approach.

3.1 Methodology

We have built the basic analyses described in Section 2 using the Open64 compilation infrastructure. Our implementation takes an input source program file and focuses on the computationally generate performance expectation metrics for various combinations of architectural elements. We also applied manual unrolling to the significant loops in the kernel code as a way to compare the expected performance of different code variants given the potential increase in instruction-level parallelism.

3.2 The Kernel Code

The computationally intensive section of the CG code is located in the *matvec* subroutine. This basic block, executed at each iteration of the loop, is the "core" of the computation, with the critical computation being an indirect access to the y array variable (see figure 3).

In the next section we review some of the experimental results for this basic block in an unmodified form, as well as manually unrolled versions in order to explore the performance impact of data dependences and the number of arithmetic and load/store units on the projected performance.

3.3 Experiments

These experiments focus on two major aspects of performance analysis: varying the number of floating point units and varying the number of load/store units. The main goal of these experiments is to understand which aspect of the computation is limiting the performance of its execution, i.e. if the computation is memory- or performance bound and how many units should be allocated to its execution in the most profitable fashion. For computations with high-latency operations such a divisions, we can easily see expected performance when such operations are pipelined versus when they are not.

Table 2 depicts the latencies of the individual operations used in our approach. In this instance, the latencies were selected to closely match that of a MIPS R10000 processor [12]. However, these latencies can be parameterized to reflect any system - real or imagined - allowing for architecture exploration that would otherwise be difficult to accomplish without cycle-level simulation. Using these parameters, the processor designer can easily "implement" any type of specialized hardware she wants (e.g. scatter-gather, very fast memory access, pipelined divides, etc.).

In order to validate our approach, we ran the CG code on a MIPS R10000 machine. Using the Performance Application Programming Interface (PAPI) [11], we were able to instrument the core of CG and determine a per-iteration cycle count of the code in the loop nest given in figure 3. Because of the sampling nature of hardware performance counters, two measurements were taken. The first sampled outside the *matvec* function call and dividing by the total number of iterations of the inner loop (all iterations). The second sampled inside the outer loop. Results for both optimized (compiler switch -O3) and non-optimized versions are shown in Figure 4. With operation latencies similar to the R10000 architecture as shown in Table 2, our predicted performance is consistent with the observed performance of the non-optimized version of the code.

These results show the programmer that regardless of a particular architectural configuration, performance is likely to be bounded by memory latency. That allows the programmer to select transformations that better mask memory latency, and informs the architecture designer as to which architectural features to choose.

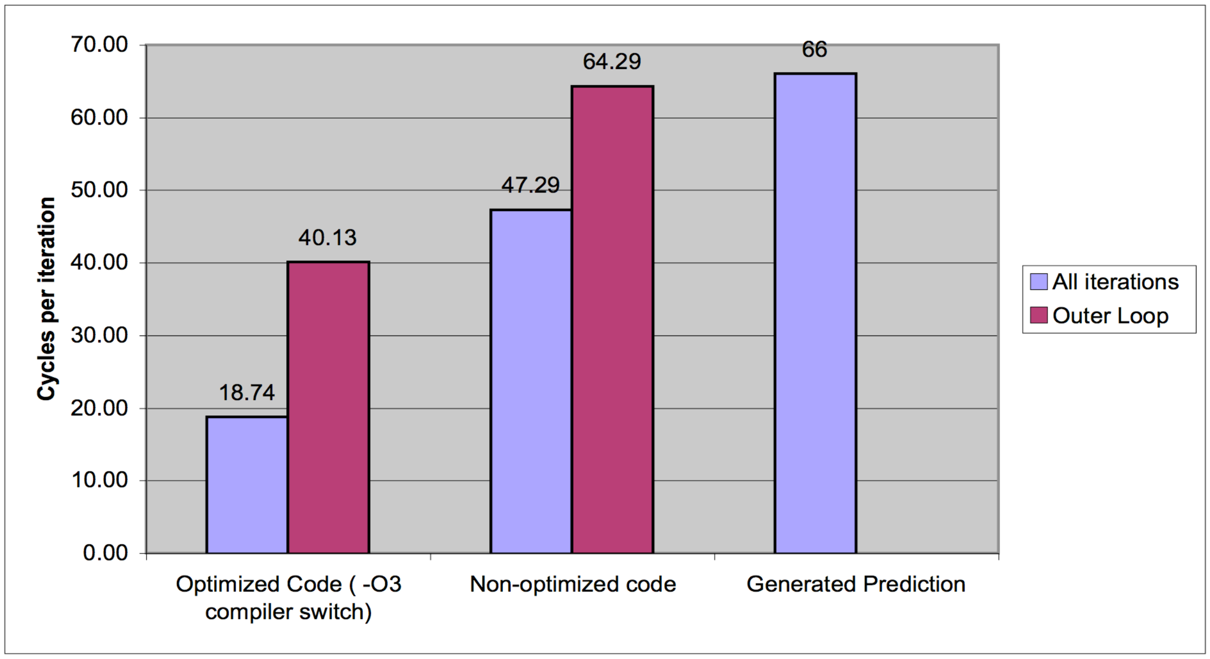

Figure 4. Validation results of CG on a MIPS R10000 machine

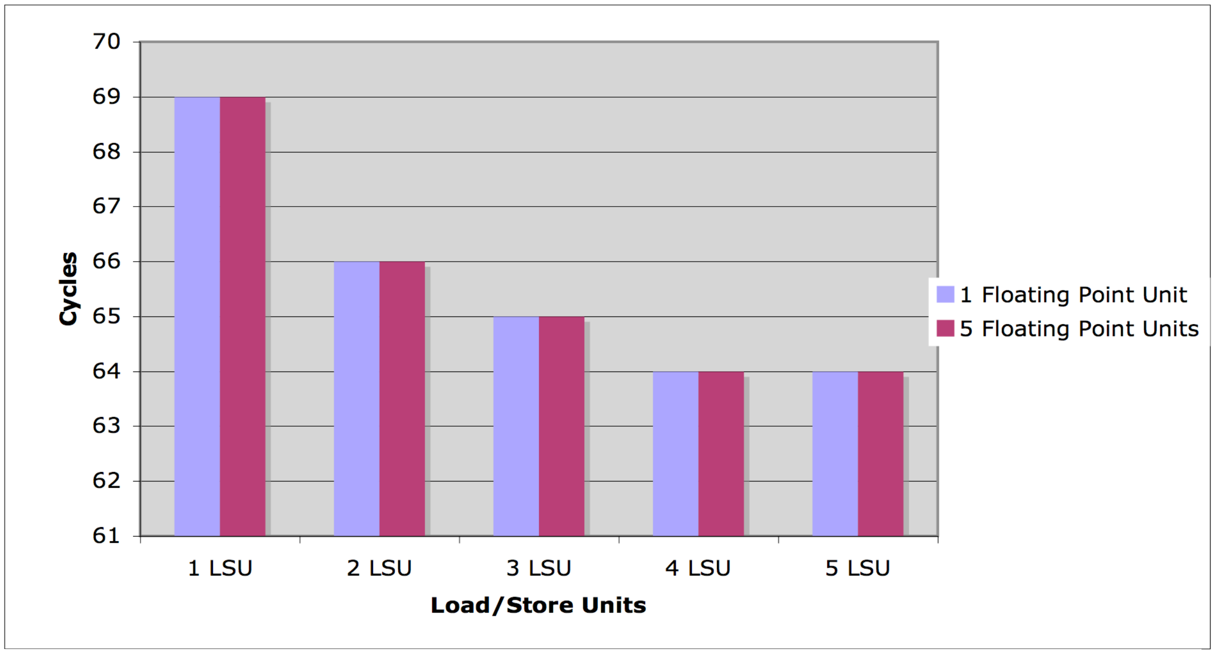

Figure 5. Cycle time for an iteration of CG with varying architectural configurations

3.4 Discussion and Future Work

The architectural model developed in the current implementation is rather simple (but not simplistic) in several respects. It assumes a zero overhead instruction scheduling. This is clearly not the case although pipelining execution techniques can emulate this aspect. It does not yet take into account advanced execution techniques such as software pipelining and multi-threading. Ignoring these techniques and compiler optimizations clearly leads to quantitative results that might differ, perhaps substantially, from current high-end machines.

Nevertheless this approach allows the development of quantitative architectural performance trends and hence allows architecture designers to make informed decisions about how to most

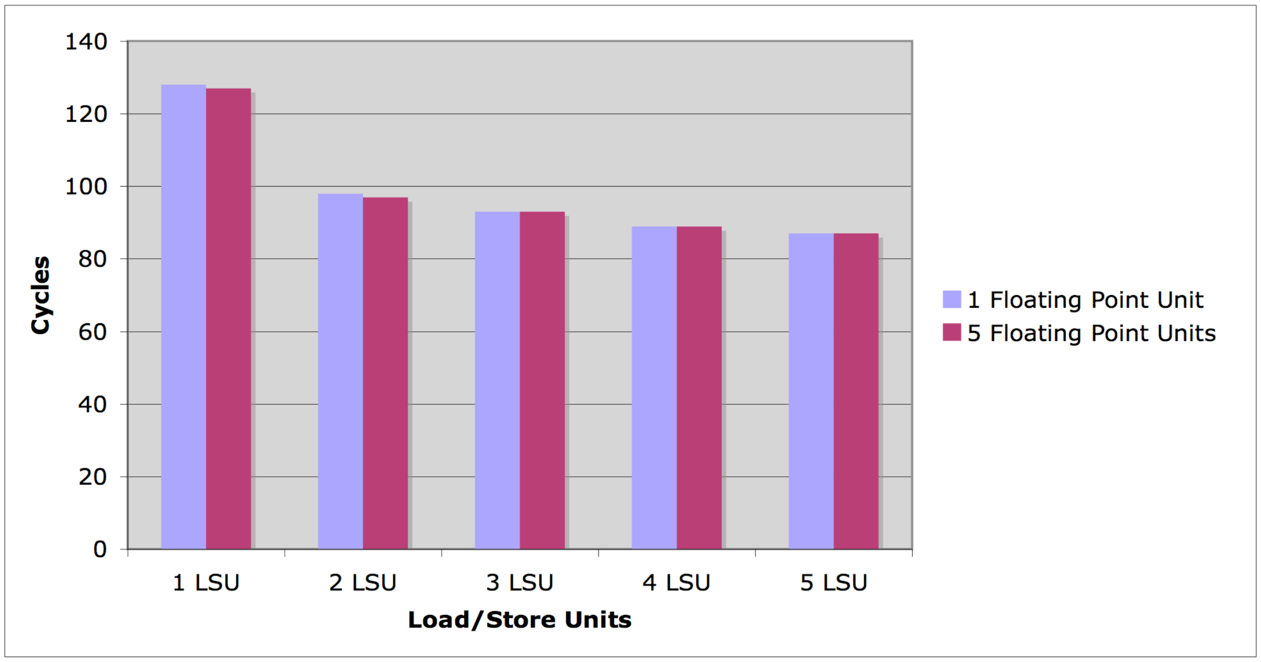

Figure 6. Cycle time for an iteration of the core of CG, unrolled 8 times, with various architectural configurations

efficiently allocate transistors. In the above case, a determination could be made between the complexity and power consumption (for example) of having more load/store units versus increasing bandwidth to memory. This information also allows developers to predict what the performance trend increases will be on a proposed "future" machine for a given code.

Future work will focus on the following: 1) A more comprehensive validation, using other codes, such as UMT2K [2] as well as other architectures, such as the IBM Power5. 2) Refinements to the register allocation and scheduling techniques. 3) A more accurate data-reuse model, such as given in [14]. 4) Using performance skeletons [13] and code instrumentation to automate the gathering of run-time data such as loop bounds and trip counts. These enhancements are currently being developed.

4. Conclusion

We have described an Open64-based system for performance prediction and architecture sensitivity analysis. Using source-level program analysis and scheduling techniques, the approach presented here provides a very fast qualitative analysis of the performance of a given kernel code. We have experimented with a real scientific code engineers and scientists use in practice. The results yield important qualitative performance sensitivity information that can be used when allocating computing resources to the computation in a judiciously fashion for maximum resource efficiency and/or help guide the application of compiler transformations such as loop unrolling.

[1] T. Mowry, Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching, ACM Transactions on Computer Systems, 16(1), pp. 55-92, Feb. 1998.

[2] The ASCII Purple Benckmark Codes (https://asc.llnl.gov/i/platforms/purple/rfp/benchmarks/).

[3] The Open64 Compiler and Tools. (http://www.open64.net)

[4] U. Banerjee, R. Eigenmann, A. Nicolau and D. Padua, Automatic Program Parallelization, In Proc. of the IEEE, 1993. '

[5] A.Snavely, L.Carrington, N.Wolter, J.Labarta, R.Badia, A.Purkayastha, "A Framework for Application Performance Modeling and Prediction", In Proc. of the 2002 ACM/IEEE SuperComputing Conference (SC'02), 2002.

[6] R.H. Saavedra and A.J. Smith, "Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times", IEEE Transactions on Computers, vol. 44:10

[7] D.J. Kerbyson, A. Hoisie, and H.J. Wasserman, "Modeling the Performance of Large-Scale Systems", Keynote paper, UK Performance Engineering Workshop (UKPEW03), July, 2003..

[8] D. Bailey, J. Barton, T. Lasinski, H. Simon, "The NAS parallel benchmarks", International Journal of Supercomputer Applications, 1991.

[9] SPEC, http://www.spec.org/

[10] P. Dinz and T. Khrisna, "A Compiler-guided Instrumentation for Application Behavior Understanding", in CTWatch Quarterly (online journal at www.ctwatch.org),, Vol.2, Number 4B, p.27-34, Nov. 2006.

[11] Performance Application Programming Interface (PAPI), http://icl.cs.utk.edu/papi/

[12] Yeager, K. C. 1996. The MIPS R10000 Superscalar Microprocessor. IEEE Micro 16, 2 (Apr. 1996), 28-40. DOI= http://dx.doi.org/10.1109/40.491460

[13] Sodhi, S. and Subhlok, J. 2005. Automatic Construction and Evaluation of Performance Skeletons. In Proceedings of the 19th IEEE international Parallel and Distributed Processing Symposium (Ipdps'05) - Papers - Volume 01 (April 04 - 08, 2005). IPDPS. IEEE Computer Society, Washington, DC, 88.1. DOI= http://dx.doi.org/10.1109/IPDPS.2005.117

[14] Wolf, M. E. 1992 Improving Locality and Parallelism in Nested Loops. Doctoral Thesis. UMI Order Number: UMI Order No. GAX93-02340., Stanford University.

[15] Keith D. Cooper and Linda Torczon. Engineering a Compiler. Morgan-Kaufmann Publishers, 2003.T. Mowry, Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching, ACM Transactions on Computer Systems, 16(1), pp. 55-92, Feb. 1998.